Any application, service, or software that consists of multiple parts communicating with each other, after reaching moderate complexity, requires some form of event/message management. Such event/message management platforms may come in a message queue(MQ) such as RabbitMQ or a message broker platform like Apache Kafka.

Both RabbitMQ and Apache Kafka use asynchronous messaging to pass information from producers to consumers. The producer can deliver a message, and if the consumer is at max capacity, down, or otherwise not ready, then the message is stored. Storing messages can allow producers and consumers to be active at different times, thus reducing coupling and increasing the system's fault tolerance.

Note: Throughout this article, the usage of events and messages are interchangeable and denote the same things, that is, technical "events". When talking about abstract concepts, events are called messages for abstraction, the technically correct term here would be events.

Table Of Contents

- What is RabbitMQ?
  - Features of RabbitMQ
- What is Kafka?
  - Features of Kafka
- Difference Between RabbitMQ and Kafka
  - Pull vs Push Approach
  - Effects of Differences on Architecture and Connections
- Conclusion
- When to use RabitMQ?
- When to use Apache Kafka?
- FAQs

What is RabbitMQ?

RabbitMQ is an open-source general-purpose *message-broker software* – an intermediary for messaging, that initially implemented the Advanced Message Queuing Protocol; it allows for various protocol extensions via a plug-in architecture. RabbitMQ can deal with some high-throughput use cases, such as online transactions or payment processing. It is generally used to handle background and cron jobs or as a message broker between microservices.

Features of RabbitMQ

RabbitMQ is lightweight and easy to deploy in-palace or in the cloud. It supports multiple messaging protocols and supports deployment in distributed and intermixed configurations to meet high-scale and high-availability requirements. RabbitMQ supports a few protocols in the form of plug-ins like AMQP 0-9-1 and *extensions*, STOMP, MQTT, AMQP 1.0, HTTP and WebSockets. While HTTP is not strictly a messaging protocol, RabbitMQ can transmit messages over HTTP in a few ways. The mainstream features of RabitMQ are Reliability and Performance, Flexible Routing, Clustering, Federation, Highly Available Queues, Multi-protocol, Many Clients, Management UI, Tracing and its versatile Plugin System.

Reliability and Performance
With RabbitMQ, you have the option to trade off performance with reliability. You can also trade for performance by sacrificing persistence, delivery acknowledgements, publisher confirms, and high availability.

Flexible Routing
Messages are routed through exchanges before arriving at queues, thus making complex routing possible. RabbitMQ features several built-in exchange types for the typical routing logic. You can even bind exchanges together or write a custom exchange type as a plugin for even more complex routing.

Clustering
You can form a single broker from multiple RabbitMQ servers on a local network by clustering them together.

Federation
RabbitMQ offers a federation model for servers that need more loosely connected than

clustering allows, but such connections are also unreliable.

Highly Available Queues
To increase reliability by ensuring that your messages are safe in the event of hardware failure, you can mirror Queues across several machines in a cluster; this trades for performance.

Multi-protocol
RabbitMQ supports messaging over a variety of messaging protocols like STOMP, MQTT and AMQP.

Many Clients
RabbitMQ has clients for almost any language you may use.

Management UI
RabbitMQ comes with an easy-to-use management UI that allows you to monitor and control every aspect of your message broker.

Tracing
RabbitMQ offers trace support to let you debug and discover what's happening if your messaging system misbehaves.

Plugin System
RabbitMQ offers a variety of ready to use plugins that extend it in different ways. You can also write your custom plugins.

What is Kafka?

Unlike RabbitMQ, Kafka is a framework implementation of a software bus using a pub-sub model of stream-processing, which means it is a distributed publish-subscribe messaging system. However, instead of using data packets, it uses a data stream to deliver the messages. These data streams are suitable for both offline and online message consumption.

Kafka is an open-source software platform developed by the Apache Software Foundation written in Scala and Java. The project aims to provide a unified, high-throughput, low-latency platform for real-time handling data feeds—many companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

Features of Kafka

Kafka aims to provide solutions for large scale event-driven systems. The top features of Kafka are Scalability, High-Volume, Data Transformations, Fault Tolerance, Reliability, Durability, Performance, Zero Downtime, Extensibility and Replication. In other words, everything a large scale event management platform would require. For each feature, Kafka implements them in the following way:

High-Volume
Kafka works with a considerable volume of data in the data streams.

Scalability
Kafka scales easily without downtime by handling scalability in all four dimensions, i.e. event producers, event connectors, event processors and event consumers.

Fault Tolerance
Kafka connector can handle failures with three strategies summarised as fast-fail, ignore and re-queue(sends to another topic).

Durability
Kafka uses a distributed commit log, which means no cascade failure and messages are persisted on a disk as fast as possible. These features make it very durable, as there can not be a single point of failure.

Performance
Kafka has high throughput for both publishing and subscribing to messages. It maintains stable performance even if many Terabytes of messages are stored.

Zero Downtime
Using replication-factor > 1 for brokers, we can have zero downtime and data loss.

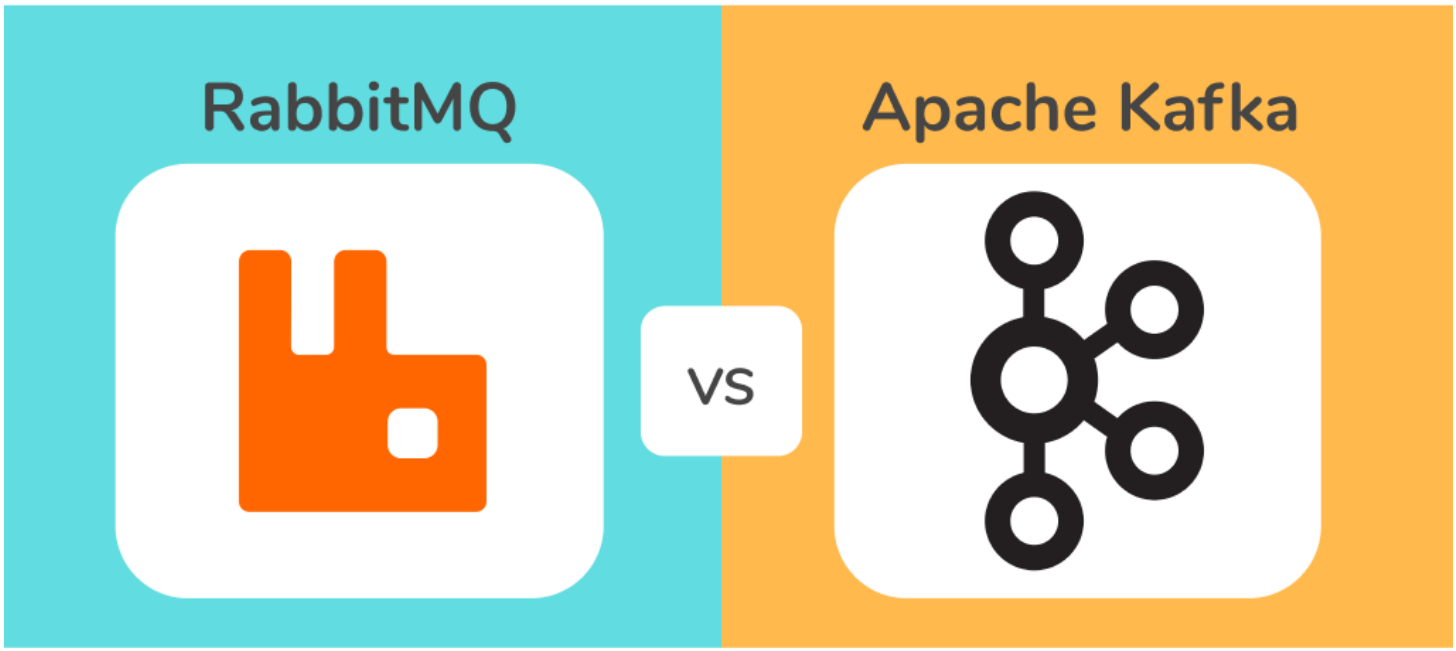
Data Transformations
Kafka allows for deriving new data streams using the existing data streams from producers.

Extensibility
Allows multiple ways for applications to plugin and make use of Kafka. Also, it has provisions for new connectors that you can write as needed.

Replication
It can replicate the events in a broker by using ingest pipelines.

Reliability
The distributed, partitioned, replicated, and fault-tolerant nature of Kafka makes it very Reliable.

Difference Between RabbitMQ and Kafka



Both RabbitMQ and Kafka serve the same purpose, and they are event handling systems that are open-source and commercially-supported pub/sub systems, readily adopted by enterprises. But these both serve similar roles but in different capacities. As will be apparent from the differences, both serve separate use cases with only minor overlaps.

Pull vs Push Approach

Apache Kafka: Pull-based approach

Kafka uses a pull-based model with *a smart consumer*, which means that the consumer has to request batches of messages from a specific offset. Kafka permits long-pooling (*the ability to configure the time interval a Kafka producer sends you another batch of events*), allowing different consumers to consume events at a different pace. It also prevents tight loops when there is no message past the offset.

When there are no contending consumers, the Kafka log preserves the order of messages in a single partition, making it necessary to use a pull model. Long-pooling also allows users to leverage the batching of messages for effective message delivery and higher throughput.

RabbitMQ: Push-based approach

RabbitMQ uses a push-based model with a smart producer, which means the producer decides when to push data. A prefetch limit is defined on the consumer to stop the producer from overwhelming consumers. Such a push-based approach is suited for low latency messaging.

The push model aims to parallelize the workload evenly between different consumers by distributing messages individually and quickly. Because of this, messages are processed only approximately in the order in which they arrived in the queue. The order is approximate and not exact since some messages may be processed faster than others.

The other significant differences are architecture and how messages are processed, which are listed below.

Differences between RabbitMQ and Apache Kakka in a tabular form:

Parameter	RabbitMQ	Kafka
Performance	Up to 10K messages per second	Up to 1 million messages per second
Data Type	Transactional	Operational
Synchronicity of messages	Can be synchronous/asynchronous	Durable message store that can replay messages
Topology	Exchange type: Direct, Fan out, Topic, Header-based	Publish/subscribe based
Payload Size	No constraints	Default 1MB limit
Usage Cases	Simple use cases	Massive data/high throughput cases
Data Flow	Distinct bounded data packets in the form of messages	Unbounded continuous data in the form of key-value pairs.
Data Unit	Message	Continuous stream
Data Tracking	Broker/Publisher keeps track of message status (read/unread)	Broker/Publisher keeps only unread messages; it doesn't retain sent messages.
Broker/Publisher Type	Smart	Dumb

Parameter	RabbitMQ	Kafka
Consumer Type	Dumb	Smart
Routing messages	Complex routing is possible based on event types	Complex routing is not possible; however, we can subscribe to individual topics.
Topology	Exchange queue topology	publish/subscribe topology
Message delivery system	Message pushed to specific queues	Pull based model; consumer pulls messages as required
Message management	Prioritize messages	Order/Retain/Guarantee messages
Message Retention	Acknowledgement based	Policy-based (e.g., ten days)
Event storage structure	Queue	Logs
Consumer Queues	Decoupled Consumer queues	Coupled consumer partition/groups

Effects of Differences on Architecture and Connections

Let's elaborate on the final listed difference of consumer queues. Suppose we have:

Application 1 is a Producer, and it produces: Event 1, Event 2
Application 2 is a Consumer, and it consumes: Event 1, Event 2
Application 3 is a Consumer, and it consumes: Event 1 only

When solving this through RabbitMQ, we will create two consumer queues, one for each consuming app. This way, we decouple our routing logic from our consumer logic as we don't need to specify this to the consumers.

Solving the same use case with Kafka, we will have one partition each for Event 1 and 2. App 2 will subscribe to both partitions, and App 3 has to subscribe only to the second partition. Kafka requires us to plan partitions ahead of time, requiring a certain amount of foresight into the use case/situation.

In case both events 1 and 2 are placed into a single partition, both apps subscribe to the partition, but app 3 needs to filter out/ignore the events of type event 1, which mixes routing logic into the consumer logic of app 3. Furthermore, Kafka gets complicated when you consume events from different topics in the same application; the same is much simpler in RabbitMQ.

Conclusion

RabbitMQ is better for applications where the architecture of the application is unknown, and it develops and evolves with the problem statement and the solution. RabbitMQ is much more flexible and easy to use in these circumstances as compared to Kafka. But once the application matures and there is a requirement for scaling, large throughput, reliability, robustness and replayability of messages, then RabbitMQ become a bottleneck, and it's better to switch to Kafka.

When to use RabitMQ?

Use RabbitMQ when:

- RabbitMQ can come in handy when you don't need the feature to replay messages on a topic. RabbitMQ cannot replay events, but sent messages are still stored; therefore, you can leverage the producer to replay the message.
- There is no clear picture of the end to end architecture. RabbitMQ is more flexible than Kafka, and it can cope with any changes in structure using its flexible routing capabilities.
- When adding consumers dynamically, RabbitMQ does not require you to change the publisher.
- RabbitMQ is language-agnostic, which means that you can create microservice in different languages, and RabbitMQ will still support such an architecture. RabbitMQ provides more language integrations than Kafka.
- Use RabbitMQ when your application needs to support legacy protocols such as AMQP 1.0, AMQP 0-9-1, STOMP, MQTT.
- Use RabbitMQ when you need some guaranteed handling of every message or some consistent behaviour against every message.
- Use RabbitMQ when there are complex point-to-point interactions(requests and responses) between many microservices that publish/subscribe.

When to use Apache Kafka?

- When there is a need to replay messages, the consumer can directly replay them. Replay makes it so that you don't lose any events if there is a bug in the consumer or the consumer is overloaded or otherwise not ready. You can simply fix the issue, bring the consumer to a *ready state*, and replay the messages.
- When you need to consume messages very quickly, Kafka should be your go-to event handling platform.
- When your application has a High throughput(100K/sec events or more), i.e. application has to process a large volume of messages.
- When the event stream needs to process data in multi-stage pipelines, the pipelines can generate graphs of the real-time data flows, thus providing real-time monitoring of traffic in the pipelines.

FAQs

Which is better, RabbitMQ or Kafka?
Objectively, in terms of performance and reliability, Kafka is better than RabbitMQ, but RabbitMQ is more flexible and easier to use. Both are suitable for specific use cases.

Does Kafka use RabbitMQ?
No, Kafka does not use RabbitMQ within its implementation or otherwise in plugin form. It is possible to stream the messages from RabbitMQ into Kafka. Such a use case often appears because RabbitMQ is already in use, and it's much easier to stream the messages from it into Kafka than to re-do all the connections for the existing applications.

Is RabbitMQ push or pull?
RabbitMQ pushes the messages into queues based on a smart publisher – dumb consumer model as given in the difference table above. Having a smart publisher allows for better control and ease in routing messages.

Is RabbitMQ fast?
Yes, RabbitMQ is fast, but in terms of handling large amounts of events, it is not as fast as Kafka. As mentioned earlier in this article, there is a trade-off between performance with reliability, including persistence, delivery acknowledgements, publisher confirms, and high availability. Meaning you can have speed, but you will sacrifice reliability and robustness for it; this is also true for Kafka but to a much lesser degree.